

Satogaeri

Markus Leitner

Contents

1	Introduction	1
2	Satogaeri	1
2.1	Rules	1
3	Satisfiability Modulo Theories	2
3.1	Eager SMT	3
3.2	Lazy SMT	3
3.3	SMT-Library	3
3.4	Logics	3
3.5	CVC4	4
4	The Solver	4
4.1	Presetting	4
4.2	Declaring Functions	5
4.3	Country Rules	6
4.4	Circle Rules	6
4.5	The solution	7
5	The Generator	8
5.1	Country generation	8
5.2	Circle generation	9
5.3	Verifying uniqueness	10
6	Drawing	10
7	Statistics	10
8	JavaFX	10
9	Conclusion	10

1 Introduction

2 Satogaeri

Satogaeri is a logic puzzle published by the Japanese company Nikoli Co.m Ltd. in 2002. Satogaeri only made a booklet appearance in Nikoli Vol. 99, Vol. 100 and Vol. 101. However it got revived on nikoli.com in 2013.

2.1 Rules

The official rules from Nikoi state:

1. The areas enclosed by bold lines, are called "Countries." Move the circles, vertically or horizontally, so each country contains only one circle.

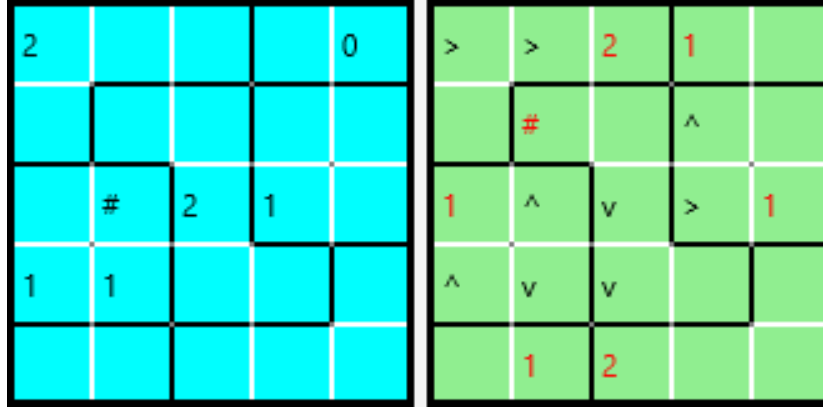


Fig. 1: Small sample puzzle with its solution

2. The numbers in the circles indicate how many cells they have to pass through. Circles without numbers may move any distance, but some of them do not move.
3. The circles cannot cross the tracks of other circles and cannot go over other circles.

And additionally the 4th rule, which counts for every logic puzzle:

- 4 A puzzle only has one solution.

Handcrafted puzzles tend to have a mirrored country-layout either vertically, horizontally or both as can be seen in Figure 1. Numberless Circles are here referred to as '#'.

3 Satisfiability Modulo Theories

Before we can go on and talk about how to solve Satogaeri problems we first have to look into Satisfiability Modulo Theories (SMT) for this was used to create the solver and the generator.

SMT can be seen as an enhancement of the traditional Satisfiability (SAT) solving. In SAT solving one tries to find an interpretation of a given problem, where this interpretation is a boolean formula composed of boolean variables and expressions like AND, OR and NOT; to state a few. If this constructed formula has a configuration of its variables where the formula evaluates to 'true', then also the original problem must have a solution. A satisfying configuration of the variables often gives a good clue on how to solve the original problem. To state an example: the formula

$$(a \vee b \vee (\neg c)) \iff ((\neg a) \wedge c)$$

is satisfied with the assignment a is 'false', b and c are 'true'. Therefore the originating problem of this formula must be solvable too. SMT goes one step beyond that by replacing some boolean variables with predicates. A predicate is basically a binary-valued function of non-binary variables, which allows us to

have function symbols with different arities. Now we can express formulas in first-order logic like

$$x > y \wedge 3x + 4 \leq 4y$$

where a natural number can be seen as a function symbol with arity 0, a so called constant. There are many different approaches of implementing SMT solvers, but the two major ones are the *eager* and the *lazy* approach.

3.1 Eager SMT

In the *eager* approach one tries to convert the given formula into an equisatisfiable propositional formula with all the additional needed constraints of the originating problem. With this constraints the search-space should be reduced so that a common SAT solver can solve the formula much quicker. And this is the appealing part of this approach: one can use already existing SAT solver to finally check the satisfiability. However the converting process into a propositional formula can be cost-intensive.[?]

3.2 Lazy SMT

The *lazy* approach consists of two parts. The first part are theory-specific solvers (\mathcal{T} -solvers) to handle conjunctions of literals, which are embedded again into a SAT solver. The advantage of incorporating \mathcal{T} -solvers is that one can use a specific algorithm and data-structure which fits the problem at hand best. This should lead to a better performance.[?]

3.3 SMT-Library

The SMT-Library (SMT-Lib) was found in 2003 to provide common standards and a library of benchmarks for diverse SMT-Solvers. With this standards and benchmarks evaluation and comparison of SMT-Systems should be made easier to give the research a competitive touch, which ultimately should result in advancing the state of the art in the SMT-Field. They even organize an annually SMT-Solver competition (SMT-COMP). SMT-Lib was greatly useful for this thesis by providing an easily understandable syntax which made many SMT-Solver accessible, like Boolector, CVC4 and MathSAT 5 to state a few. On their web-site¹ one can find a long list of Solver supporting SMT-LIB and a separate list of these Solver still under active development.

3.4 Logics

SMT-LIB supports a huge variety of different logics. The advantage here is to be able to apply more specialized and therefore more efficient algorithms. The meaning of the logics in Figure 2 is letter coded. For example:

1. QF stands for the restriction to quantifier free formulas
2. IA stands for the theory Ints (Integer Arithmetic)
3. L before IA, RA, or IRA stands for the linear fragment of those arithmetics

¹ www.SMT-LIB.org

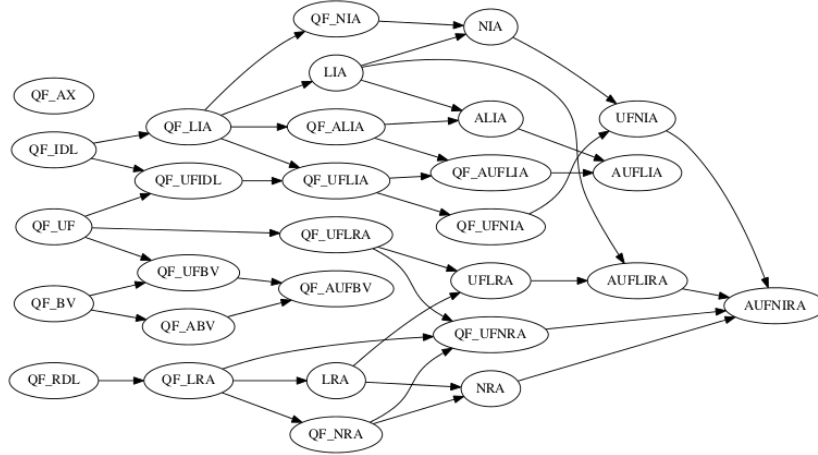


Fig. 2: A link from a logic L1 to a logic L2 means that every formula of L1 is also a formula of L2

So QF_LIA is an unquantified linear integer arithmetic. In essence, Boolean combinations of inequations between linear polynomials over integer variables. The whole explanation can be found on the SMT-Lib web-site² in the section Logics.

3.5 CVC4

The SMT-Solver used in this project is the Cooperating Validity Checker 4 (CVC4), currently the most novel successor of the Stanford Validity Checker which was found in 1996. CVC4 is a joined project of the New York University and the University of Iowa. Supported features are inter alia: several built-in base theories, support of quantifiers and a model generation ability. The latter is a very important requirement for this project at hand. Furthermore CVC4 has its own wiki-web-page³ with useful tutorials, user manual and an developer section.

4 The Solver

Now we will discuss how to represent a Satogaeri puzzle in SMT and what are the needed inputs for the CVC4 solver. Because we use the SMT-Lib and its generalized input syntax the CVC4 solver could be replaced by any other solver supporting the SMT-Lib. For illustration purposes we will now manually create a SMT representation of the Satogaeri puzzle of Figure 3.

4.1 Presetting

Before starting up the CVC4 solver we need to ensure some presetting which is mandatory for our purpose. The parameter additionally given to the program via the command-line are the following:

² www.SMT-LIB.org

³ <http://cvc4.cs.nyu.edu/wiki>

#		
1		0

Fig. 3: Trivial Satogaeri puzzle

```
—lang smt -m —statistics
```

-lang smt enables the use of the SMT-Lib syntax other than the regular CVC4 syntax.

-m enables the generation of a model which means once our problem is satisfiable the solver also generates a satisfying assignment for all variables to give us an example model in which the problem has a solution.

-statistics prints a long list of statistics if our problem was satisfiable. We will use these statistics to decide how 'difficult' the given Satogaeri puzzle was. More about that later in section "".

4.2 Declaring Functions

Now that CVC4 is started with the proper presetting we can start our representation of the puzzle in Figure 3.

First thing to do is to define the logic to be used.

```
(set-logic QF_LIA)
```

In our case it is the unquantified linear integer arithmetic. Simple because we do not need quantifiers nor anything other than simple integer functions. By restricting the search-space for the solver it should be faster in find a solution.

Finally we can start our SMT-interpretation of the Satogaeri puzzle. First of all we define every cell of the puzzle as a constant integer. One can think of the puzzle as an 2-by-3 matrix filled with yet unknown numbers.

```
(declare-fun f1-1 () Int)
(declare-fun f1-2 () Int)
(declare-fun f1-3 () Int)
(declare-fun f2-1 () Int)
(declare-fun f2-2 () Int)
(declare-fun f2-3 () Int)
```

This lines introduce a new function with the name e.g. 'f2-3' which has no arguments (empty brackets) and returns an integer. 'f2-3' in our case represents the field in row 2 and column 3. The collection of these functions will be our matrix, to be more precise its the matrix where we will define our 'country' restrictions of the Satogaeri rules, referred to as country-matrix from here on.

Now we will create a second 2-by-3 matrix just to model the restrictions of the circles.

```
(declare-fun c1-1 () Int)
(declare-fun c1-2 () Int)
(declare-fun c1-3 () Int)
(declare-fun c2-1 () Int)
(declare-fun c2-2 () Int)
(declare-fun c2-3 () Int)
```

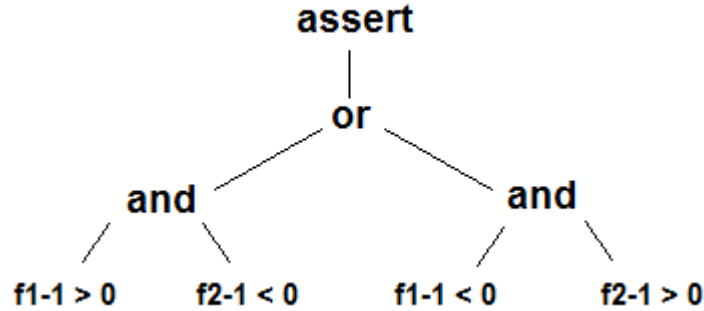


Fig. 4: Tree-structure representation of a SMT line

Same principle as before only that the 'c' in 'c2-3' stands for circle. One can think of it as a circle-matrix on top of an country-matrix each of those only checks for specific rules of the Satogaeri puzzle.

4.3 Country Rules

We declared a country-matrix but there are no restrictions on it whatsoever. Next up is the definition of the rule: 'one or several fields combined result in a country. Every country only can be occupied by one circle.' For example 'f1-1' and 'f2-1' combined are one country. That means once 'f1-1' is occupied 'f2-1' is not allowed accommodate any circles and vice versa. Later on we will define that a circle is a positive natural number, where as no circle is a negative number. In the SMT-syntax it will look like this

```
( assert ( or ( and ( > f1-1 0 ) ( < f2-1 0 ) ) ( and ( < f1-1 0 ) ( > f2-1 0 ) ) ) )
```

Important to state is that the SMT-syntax consists only of prefix notations. The simplest way to look at this line is as a tree-structure. (Figure 4)

assert will make sure that in the end there is an assignment to these fields that this line will become 'true' otherwise the problem is unsatisfiable. The *or* states that at least one of its arguments must be 'true', and the *and* only yields true if all of its arguments evaluate to 'true'. Both *or* and *and* can take one or more arguments inside its brackets whereas *assert* only takes one.

This line basically translates to: either f1-1 is a positive number and f2-1 is negative or vice versa.

Now we have successfully translated rule number 1 into SMT-syntax for one country. Let's do the same with the other two countries:

```
( assert ( or ( and ( > f1-1 0 ) ( < f2-1 0 ) ) ( and ( < f1-1 0 ) ( > f2-1 0 ) ) ) )
( assert ( or ( and ( > f1-1 0 ) ( < f2-1 0 ) ) ( and ( < f1-1 0 ) ( > f2-1 0 ) ) ) )
```

And we are done with the country-matrix.

4.4 Circle Rules

First thing we do is to assign every circle with its own ID.

```
( assert ( and ( = c1-1 1 ) ( = c2-1 2 ) ( = c2-3 3 ) ) )
```

At the start of this section we defined `c1-1` as a function with no arguments which yields an integer. All we did was to define that this integer has to be 1. In other words: `c1-1` is constant 1. Now the '#'-circle at `c1-1` has the ID 1, '1'-circle at `c2-1` has the ID 2 and so on and so forth.

Our next step is to define the possible moves every circle is able to make. Let's start with the easiest one. The '0'-circle at `c2-3`. The 0 indicates that this circle is not allowed to move at all which means he will occupy the field he is already in.

```
(assert (= f2-3 c2-3))
```

We have our first interaction with the country-matrix and the circle-matrix. With this line we finally declare that function `f2-3` has to yield the same value as `c2-3` which is its ID: 3. This leads to, as we remember, that the country consisting of `f2-3` and `f1-3` is occupied and `f1-3` has to be a negative number in other words is not allowed to be occupied by any circle anymore.

Next is the '1'-circle in `c2-1`. This circle can move to 2 positions either to `c1-1` or `c2-2` because of its movement-range of 1.

```
(assert (or (and (= c1-1 c2-1) (= f1-1 c2-1))
            (and (= c2-2 c2-1) (= f2-2 c2-1))))
```

In this case we not only override the position in the country-matrix but also all the fields we pass in the circle-matrix, because rule 3 states: 'The circles cannot cross the tracks of other circles and cannot go over other circles.' By leaving the ID of every circle in the fields they passed we basically leave a track which is not allowed to be crossed. For example: the '1'-circle in `c2-1` is not allowed to move to `c1-1` because '#'-circle is already there. We made sure of this by first defining the constant `c1-1` with the ID 1 and now we said if the '1'-circle wants to occupy `f1-1` it has to leave its ID in `c1-1`, which yields a contradiction because a constant can never be both 1 and 2. Long story short: in this case only `f2-2` can be reached and occupied without violating any rules.

Last one is the '#'-circle at `c1-1`. Because of rule 2: 'Circles without numbers may move any distance, but some of them do not move.' this circle has 4 possible moves to make. either stay at `c1-1`, move to `c1-2`, move to `c1-3` or move to `c2-1`. The encoding for each possible move is devoted its own line

```
(assert (or (and (= f1-1 c1-1))
            (and (= c1-2 c1-1) (= f1-2 c1-1))
            (and (= c1-2 c1-1) (= c1-3 c1-1) (= f1-3 c1-1))
            (and (= c2-1 c1-1) (= f2-1 c1-1))))
```

Now we are done. The Satogaeri puzzle of Figure 3 is translated into SMT. Mention worthy is that the final solver is smart enough to detect simple collisions like if there is a circle in the path before hand and will not add that particular move to the SMT-syntax, this reduces the possible cases the CVC4 solver has to check.

4.5 The solution

Once we translated the puzzle into an SMT-problem we can evoke the solving procedure by simply typing

```
(check-sat)
```

Now there are 3 possible outputs the solver can give:

1. unsat: It means that there is no possible assignment to solve the problem at hand.
2. unknown: Is used when a solver did not manage to establish whether the set of assertions is satisfiable or not. However some solver will create a candidate model.
3. sat: There is at least one possible assignment which satisfies the problem.

Obviously we looking for a sat result. Once the problem is sat and we enabled `-m`, the model generating, we now can check for possible assignments of all fields in the country-matrix as well as in the circle-matrix. Our major interest lies in the country-matrix:

```
(get-value (f1 -1))
(get-value (f1 -2))
(get-value (f1 -3))
(get-value (f2 -1))
(get-value (f2 -2))
(get-value (f2 -3))
```

Fields that are not occupied by any circle will yield a negative number, most likely -1. However the occupied once will give us the ID-number of the circle which is currently occupying this field, the whole country. We could also look into the circle-matrix with this lines:

```
(get-value (c1 -1))
(get-value (c1 -2))
(get-value (c1 -3))
(get-value (c2 -1))
(get-value (c2 -2))
(get-value (c2 -3))
```

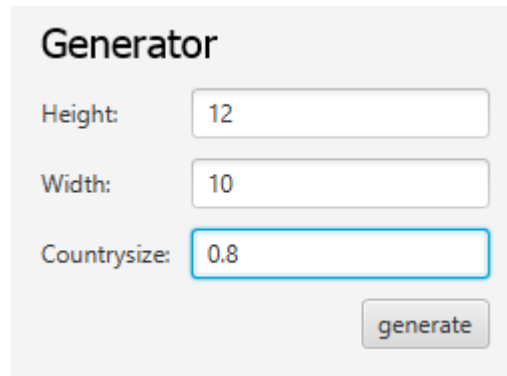
However there we will only see the path taken of every circle marked with its ID and the unneeded fields will be filled with random numbers most likely 0, but it could also be a number used as an ID for a circle which makes the circle-matrix inaccurate.

5 The Generator

The Generator should be able to generate random Satogaeri puzzles with varying size and difficulty. The user enters the width and hight of the desired puzzle and another parameter called `Countrysize` which is a double between 0.0 and 1.0. (Figure 5) The higher the value of `Countrysize` the more likely it is to get countries with a higher amount of fields. The idea was that the more smaller countries are generated the more difficult the puzzle may get. Unfortunately it contributes very little to the difficulty of the generated puzzle.

5.1 Country generation

As mentioned before the user defines the width and hight of the puzzle. Now the program will take the topmost leftmost field which is not yet involved in any



The image shows a graphical user interface titled "Generator". It contains three input fields: "Height:" with the value "12", "Width:" with the value "10", and "Countrysize:" with the value "0.8". The "Countrysize" field is highlighted with a blue border. Below these fields is a button labeled "generate".

Fig. 5: Generator graphical interface

country and will declare it as a new country. With the probability of *Countrysize* each of the four connecting fields (top, bottom, left, right) will be added to this newly created country. Next the four connecting fields of each new member have a chance of $(previous_chance)^2$ to be added too until there are no new members.

Theoretically with this algorithm it is possible to generate countries in every conceivable shape, however some general shape are more likely than others. And a very small *Countrysize*-value will end up in puzzles with a lot of countries consisting only of one field which are uninterestingly easy to solve.

5.2 Circle generation

After we know the layout of all the countries we can start adding Circles. The Generator now randomly picks one field in every country and declares it as the origin of the circle. At this point we do have the solution to the upcoming puzzle. Next step is to shuffle these circles around. Therefore the Generator randomly picks a direction in which the circle should move and calculates the distance to the next obstacle (e.g.: the boarder of the puzzle, another circle, the track of another circle). Another random number will be picked between 1 and the distance to the next obstacle and this indicates the movement of this circle. However if the obstacle is right next to the circle a new direction will be chose. If that scenario happens twice in a row, it will become a '0'-circle. This is necessary to reduce the number of boring '0'-circles but do not remove them in general.

Now everything is generated except '#'-circles. Those are not be picked randomly but must meet some requirements. A circle will become a '#'-circle if and only if:

1. from its origin point on into the moving direction it has to leave its country with the first field
2. in the opposite direction on the origin point there mus be a blockade (a circlce-trace, a circle, a boarder or a different country)
3. if its a '0'-circle it should not be possible to reach another field inside the own country

If one of these requirements is violated it will break the uniqueness of the puzzle.

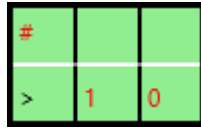


Fig. 6: Solution to trivial Satogaeri puzzle

5.3 Verifying uniqueness

Uniqueness of the solution to a puzzle is very important for the joy of play. In other words the Generator must assure that the puzzle only has one solution. This is done by simply giving the puzzle to the solver with the premiss that the already known solution is prohibited, and if the solver finds a different solution the uniqueness of the puzzle is violated.

But how to model this premiss in SMT? To answer that we will look again at the puzzle in Figure 3. But this time we need the solution to this puzzle which looks like in Figure 6. We remember that for example the '#'-circle had the ID 1. So all we have to do is to tell the origin point of '#'-circle that the ID 1 is not allowed to land there. And the same goes for the rest of the circles. In the SMT-syntax it will look like this:

```
(asser (or (not (= f1-1 1)) (not (= f2-2 2)) (not (= f2-3 3))))
```

If this line is added to our previous example of SMT-encoding the solver will return *unsat* if the given solution is the only one and will return *sat* if there is a second possible solution. Because we enabled the model generating feature we can also have a look at this second solution via the *(get-value ())* input.

6 Drawing

7 Statistics

8 JavaFX

9 Conclusion